UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 3

Inheritance and Data Structures

Date Due: May 29, 2019, 8pm **Total Marks: 120**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- **Make sure your name and student number appear at the top of every document you hand in.** These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- **Assignments must be submitted to Moodle.**
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- **Programming questions must be written in Java.**
- **Non-programming questions must be submitted as either `.txt` or `.pdf` files.** If other file types are used, you will receive a grade of zero for that question.

Version History

- **22/05/2019**: released to students

Question 1 (120 points):

Purpose: Practising Inheritance and Object Oriented Programming

Degree of Difficulty: Tricky

For this assignment, you will extend the hospital application developed in Assignment 2. You must implement the following four classes: a `Patient` class, a `Doctor` class, a `Surgeon` class and a `HospitalSystem` class. As well as modifying the `Ward` class. Each class should have a constructor, as well as the instance variables and methods specified below. For each of the classes, all the instance variables should be declared as private. Be sure to follow the specification given below, since the classes will be used in subsequent assignments in order to build a simple hospital management system. In addition, each class is to have a static main method that should be used to test the class methods following the principles of testing as discussed in class. Do not add any additional parameters to the methods, none are needed.

The four classes should extend and use the `Person`, `BasicDoctor` and `Ward` Classes that can be found in the folder titled "Provided Classes" on moodle. This ensures everyone is starting with correct code.

(a) (32 points) The `Patient` class. This class extends the `Person` class. For our purposes, a patient will have the following features:

- An integer label of the bed for the patient. A value of -1 is used if the patient has not assigned a bed
- A list to store all the patient's doctors
- `Patient(String name, int number)` A constructor that takes in the person's name and health card number. Initially, the patient should not be in a bed and have no doctors assigned to them.
- `getBedLabel()` An accessor method for the bed label
- `setBedLabel(int bedLabel)` A mutator method for the bed label
- `addDoctor(Doctor d)` A method that adds a new Doctor to the list of the patient's doctors
- `removeDoctor(String name)` A method that removes a Doctor from the list of the patient's doctors
- `hasDoctor(String name)` A method that checks to see if a Doctor with `name` is assigned to the patient. Returns true if the doctor is found, false otherwise.
- `toString()` A method that returns a string representation of all the information about the patient in a form suitable for printing. This should include the patients name, health card number, the bed label (if any) and the name of each doctor associated with the patient.
- A main method that will test all of the above features

Hint: You can use the `LinkedList<E>` class in the `java.util` library to store a list of doctors.

(b) (22 points) The `Doctor` class. This class extends the `BasicDoctor` class. For our purposes, a doctor will have the following features:

- A list of the doctor's patients
- `Doctor(String name)` A constructor that takes in the doctor's name. Initially the doctor should have no patients.
- `addPatient(Patient p)` A method that adds a patient to the doctor's list.
- `removePatient(int healthNum)` A method that removes a patient from the doctor's list.
- `hasPatient(int healthNum)` A method that checks to see if a Patient with `healthNum` is under the doctor's care. Returns true if the patient is found, false otherwise.
- `toString()` A method that returns a string representation of all the information about the doctor in a form suitable for printing

- A main method that will test all of the above features

Hint: You can use the `LinkedList<E>` class in the `java.util` library to store a list of patients.

(c) (8 points) The `Surgeon` class. This class extends the `Doctor` class. For our purposes, a surgeon will have the following features:

- `Surgeon(String name)` A constructor that takes in the surgeon's name. Initially the surgeon should have no patients.
- `toString()` A method that returns a string representation of all the information about the doctor in a form suitable for printing. This string should start off the the classifier "Surgeon: " followed by the rest of the relevant information.
- A main method that will test all of the above features

(d) (10 points) Modifications to the `Ward` class. The **provided** `Ward` class should be modified so that the array has type `Patient`, rather than `Person`. When this change is made, a number of the methods will need to be changed in order to be consistent with type `Patient` being stored in the array. In addition, the following two methods need to be added:

- `availableBeds()` A method that returns a list of the empty beds in the ward.
- `freeBed(int bedLabel)` A method that removes a Patient from a specific bed.

Make sure you update the tests in the `main` method and add new tests for the new methods.

(e) (28 points) The `HospitalSystem` class. The last class to write is one to run a simple hospital system. For our purposes, the hospital system will have the following features:

- a single Ward.
- A keyed dictionary of all the patients known to the system. Where the key is the patient's health card number.
- A keyed dictionary of all the doctors known to the system. Where the key is the doctor's name.
- `HospitalSystem()` A constructor for the class. Initially, there should be no patients and no doctors. The name of the ward and the integer labels for the first and last beds should be obtained from the user using console input and output.
- `addPatient()` A method that executes Task 2 (described below).
- `addDoctor()` A method that executes Task 3 (described below).
- `assignDoctorToPatient()` A method that executes Task 4 (described below).
- `assignBed()` A method that executes Task 6 (described below).
- `dropAssociation()` A method that executes Task 8 (described below).
- `systemState()` A method that executes Task 9 (described below).
- `toString()` A method that returns a string representation of all the information about the doctor in a form suitable for printing.
- `displayEmptyBeds()` A method stub for Task 5 (described below).
- `releasePatient()` A method stub for Task 7 (described below).
- A main method. The main method should construct a new instance of the `HospitalSystem` class. I should then display a message to the user asking the user to select a task. Once the task is selected, it should be carried out and then the system should prompt the user to select another task. It is easiest to handle task selection is by numbering the tasks and having the user enter an integer.
  1. quit

2. add a new patient to the system
3. add a new doctor to the system
4. assign a doctor to a patient (this should also add the patient into the doctor's list of patients)
5. display the empty beds of the ward
6. assign a patient a bed
7. release a patient
8. drop doctor-patient association
9. display current system state

If the user enters a `1` the system should display the current state of the system and stop prompting the user to select a new task. **Note for this assignment you do not have to implement task 5 and task 7, but there should be method stubs for these tasks that prevent the system from crashing if these tasks are selected.**

Hint: You can use the `TreeMap<K,V>` class in the `java.util` library to store a keyed dictionary. Unlike the other classes, the `HospitalSystem` class gets information from the user instead of having passing in arguments to each method. Every method above (except for `toString()` will require user input, even the constructor!)

(f) (20 points) You will need to include internal and external documentation with every class. See below for what is expected:

- Proper internal documentation includes:
  - A comment just before the class header that gives an overview of the class. For example, if the class models an entity, the comment might state what entity the class models and specify the key features. Alternatively, if the class serves as a container, state what type of items the container stores, and how these items are accessed. If the class is an interface, state what it provides an interface for and whether there is anything special about the interface. Finally, if it has a control function, what is it doing and controlling? Recall that comments for a class appear before the class and begin with `/**`.
  - A comment just before each instance variable stating what is stored in the field, again beginning with `/**`
  - A comment before each constructor and method stating what it does. Note that the comment only gives what the routine does, not how it does it. If it isn?t obvious from the code how it accomplishes its goal, comments on how it is done belong in the body of the method.
  - Single line comments as needed to improve the readability of your code. Over-use of single-line comments will result in a deduction of marks.
  - Use descriptive variable names and method names.
  - Include good use of white space, especially reasonable indentation to improve the readability of your code.
  - Hard-code all of the data for your testing. Minimize console input and output by only reporting errors(These are the same principles we applied in CMPT 145).

- External Documentation:
  - A description of how to execute your tests of the classes. What is necessary to compile your classes? What program or programs need to be run to show the execution of your program? For example, it might be necessary to compile and execute all three classes individually. On the other hand, you might have written a driver program (another class with a static main method that invokes the main methods of the three required classes; the static main method of class `A` can be invoked by `A.main(null);`). In the latter situation, we need to know the name of the driver class. This description should be very short.

- Now that we have the beginning of a complete java application, you should include a demonstration of your system running. This can be achieved by copy-and-pasting (or screen-grabbing) your console input/output that shows you invoking each task.
- The status of your assignment. What is working and what is not working? What is tested and what is not tested? If it is only partially working, the previous point should have described how to run that part or parts that work. For the part or parts not working, describe how close they are to working. For example, some of the alternatives for how close to working are (i) nothing done; (ii) designed but no code; (iii) designed and part of the code; (iv) designed and all the code but anticipate many faults; or (v) designed and all the code but with a few faults; (vi) working perfectly and thoroughly tested.
- Maintenance manual. This is information for the person or persons who must keep your system running, fix any faults, and do any upgrades. The reader of the maintenance manual is expected to have the same background and experience as yourself, but of course not having developed your system. This part of the documentation will vary from assignment to assignment. For this assignment, it is sufficient to include a UML class diagram showing all the features of each class, and the relationships (inheritance, uses and aggregation) amongst the classes (if any).
- UML diagrams should be generated with a program like UMLet or an online tool, such as `https://www.gliffy.com/uses/uml-software/`. Make sure that you use the correct arrows and arrowheads. Incorrect arrowheads will lose several marks. You can also draw out a UML diagram by hand and scan it. Just make sure it is easily readable.

Additional information

System Tasks: In each of the tasks, patients are identified by their health number, doctors by their name, and beds by their (external) integer label. When the user quits, the system should print out the system state at that time. Note that when the task is to add a new doctor, the user should be asked whether the new doctor is a Surgeon or not. If so, a Surgeon should be created. Note that as will be discussed in class soon, an object of a certain type can be assigned to a variable of an ancestor type. Thus, a Surgeon can be assigned to a Doctor variable, and a Surgeon can be placed in the dictionary of Doctors.

Commenting and Exceptions: When writing these classes, be sure to properly document each method, including `@param` and `@return` comments. Also, if a method has a precondition, specify the precondition in a `@precond` comment, and throw a runtime exception if it is not satisfied. Be sure to include a meaningful error message in the String parameter for the exception constructor. Note that these additional comments and precondition checks have already been added to the Person and BasicDoctor classes of Assignment 2, but they should be added into the Ward class, as well as the new classes. When appropriate, exceptions should be caught and handled. In particular, if a task of the system fails and as a result throws an exception, it is reasonable to print the message of the exception, and then try the next task. The system should not crash when a user enters invalid data. To achieve that, a `try-catch` will be needed for the invocation of any method to handle a task that might throw an exception. These try-catches will probably be in the system class, as it is the one that has the invocations of the methods in the entity classes. Note that you might be able to handle all situations with only a couple try blocks.

Input and Output: Note that when a value is read using Scanner, other than by nextLine(), none the characters after the value are read. Thus, a subsequent nextLine() read will read those following characters up to and including the next end-of-line character(s). Such a read often just reads end-of-line character(s) that mark the end of the line after the previous read, so that the nextLine method simply returns the empty string. It does not read the characters on the next line (which might be what was wanted), since it finds end-of-line characters first. So be careful, and use the debugger!!

Levels of Protection: In keeping with the principle of information hiding, the instance variables of a class should be private unless there is a very good reason to make them visible. This has already been done

for the classes of Assignment 2. When appropriate, methods should be supplied to access and set the fields. It is anticipated that there might be many types of specialists (descendants of Doctor). Thus, the list of patients for a doctor might be set protected, so that descendant classes can directly access this list.

What to Hand In

- The completed `Patient.java` program.
- The completed `Doctor.java` program.
- The completed `Surgeon.java` program.
- The completed `Ward.java` program.
- The completed `HospitalSystem.java` program.
- `a3q1_documentation.pdf` that contains the external documentation for each class and the application as a whole. Any documentation relating to a specific class should be easily identifiable. Include the inherited classes in the UML class diagram.

Be sure to include your name, NSID, student number and course number at the top of all documents.

Evaluation

In each class, the following evaluation scheme is used:

2 pts For each instance variable

2 pts For each method

2 pts For testing each method

3 pts For internal documentation

5 pts For external documentation